

INTRO TO PYTHON, PART 3

Functions, modules, and strings, oh my!

HOW THIS PART WILL WORK

I will give you a bunch of exercises

Do each one on your laptop

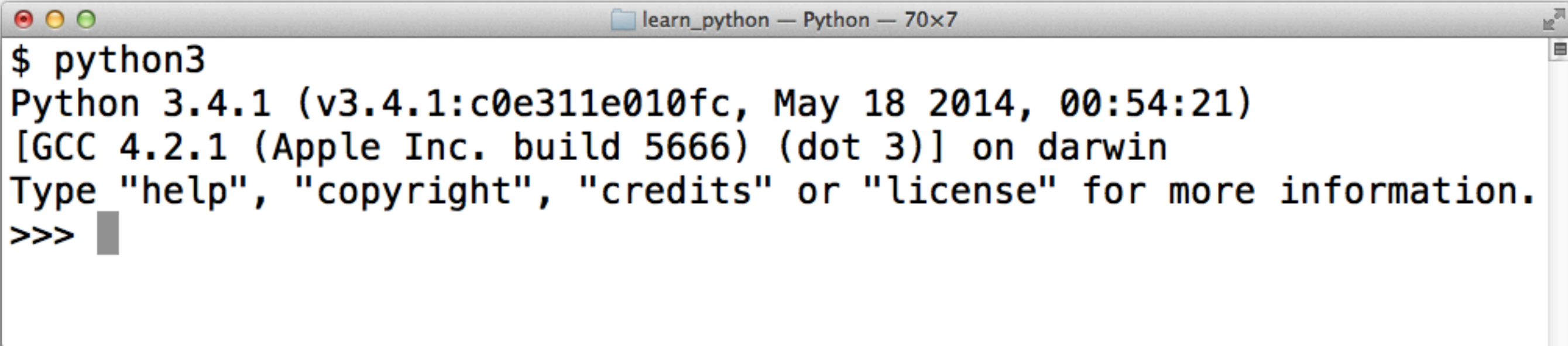
- Ask for help as needed
- Look up when you finish

Once everyone's done trying an exercise, I'll show the answer and explain it.

YOU SAW THE REPL

It's that interactive shell that you've been playing with.

Yes, the one that looks like this:

A screenshot of a macOS terminal window. The title bar at the top reads "learn_python — Python — 70x7". The terminal content shows the command "\$ python3" being executed, followed by the Python 3.4.1 startup banner: "Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)", "[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin", and "Type 'help', 'copyright', 'credits' or 'license' for more information.". The prompt ">>>" is visible at the bottom with a cursor.

```
$ python3
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

NOW, MEET PYTHON SCRIPTING

Python scripts are:

- Just text files
- With filenames that end in .py (not .txt)

A type of simple Python program

HELLO WORLD

The first example program for any programming language.

Traditionally used to illustrate to beginners the most basic syntax of a programming language.

Drumroll...here it is...

EXERCISE 0

Open a text editor

Type this:

- `print("Hello, World!")`

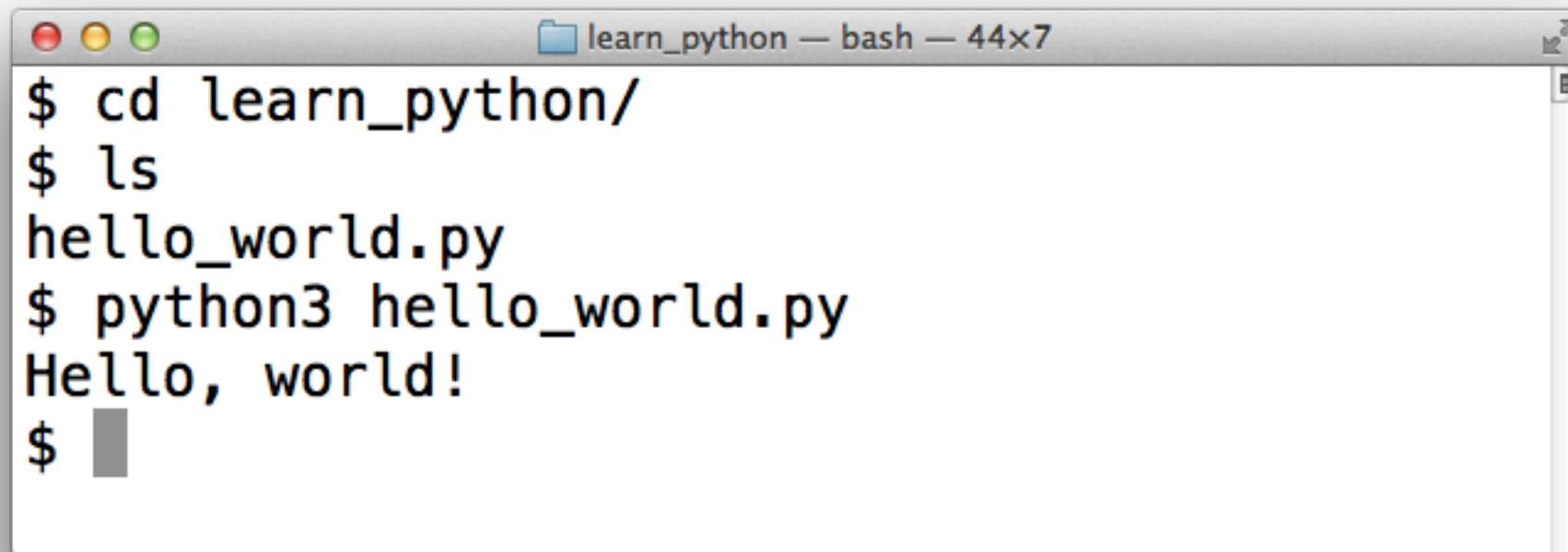
Save the file as ***hello_world.py***

- Where? Within a directory called ***learn_python***, which can be anywhere on your computer.

EXERCISE 0

Open Terminal or Powershell

Navigate to the script and run it:

A screenshot of a terminal window titled "learn_python — bash — 44x7". The terminal shows the following commands and output:

```
$ cd learn_python/  
$ ls  
hello_world.py  
$ python3 hello_world.py  
Hello, world!  
$
```

The terminal window has a standard macOS-style title bar with red, yellow, and green window control buttons on the left. The text is displayed in a monospaced font.

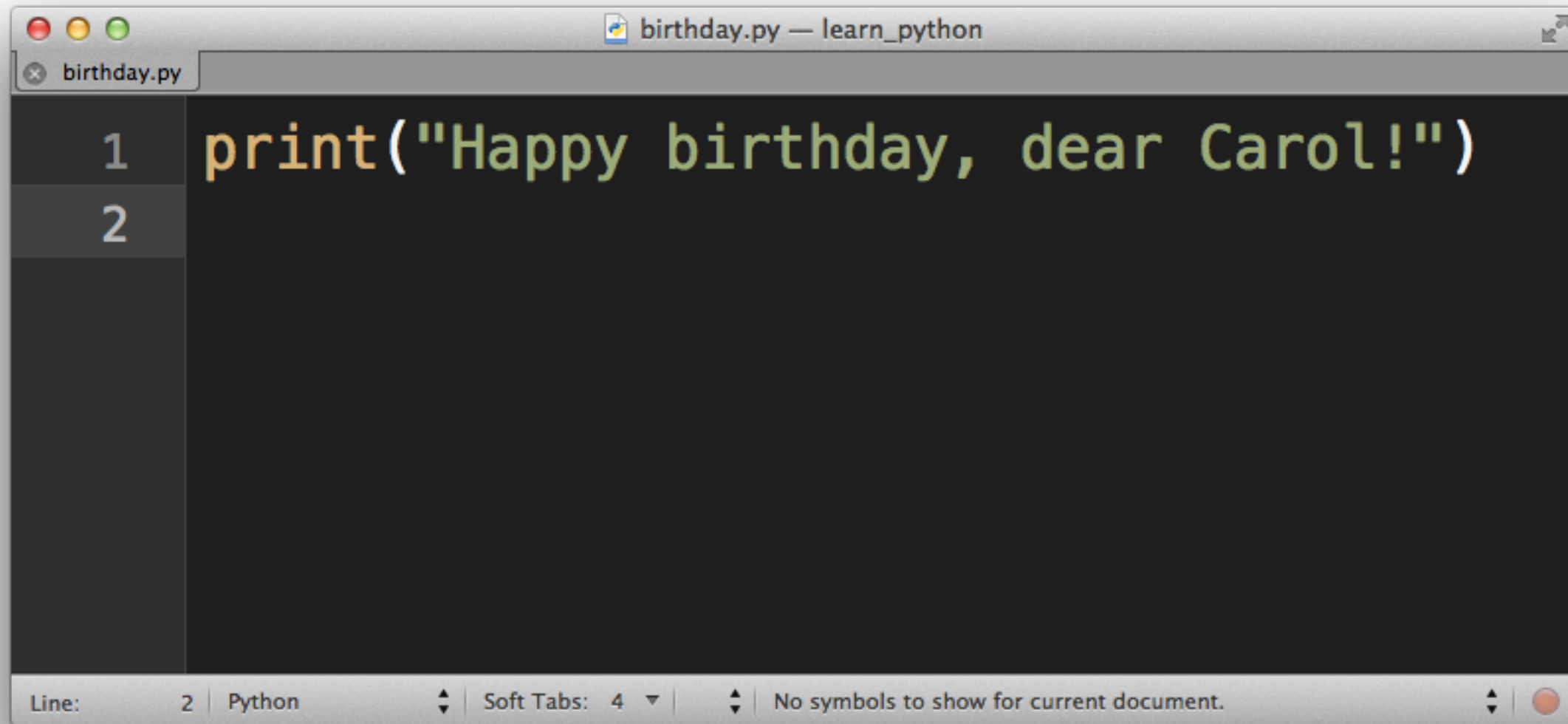
EXERCISE 1: YOUR CHALLENGE

Now write a similar program called ***birthday.py***

When you run ***birthday.py***, it should print:

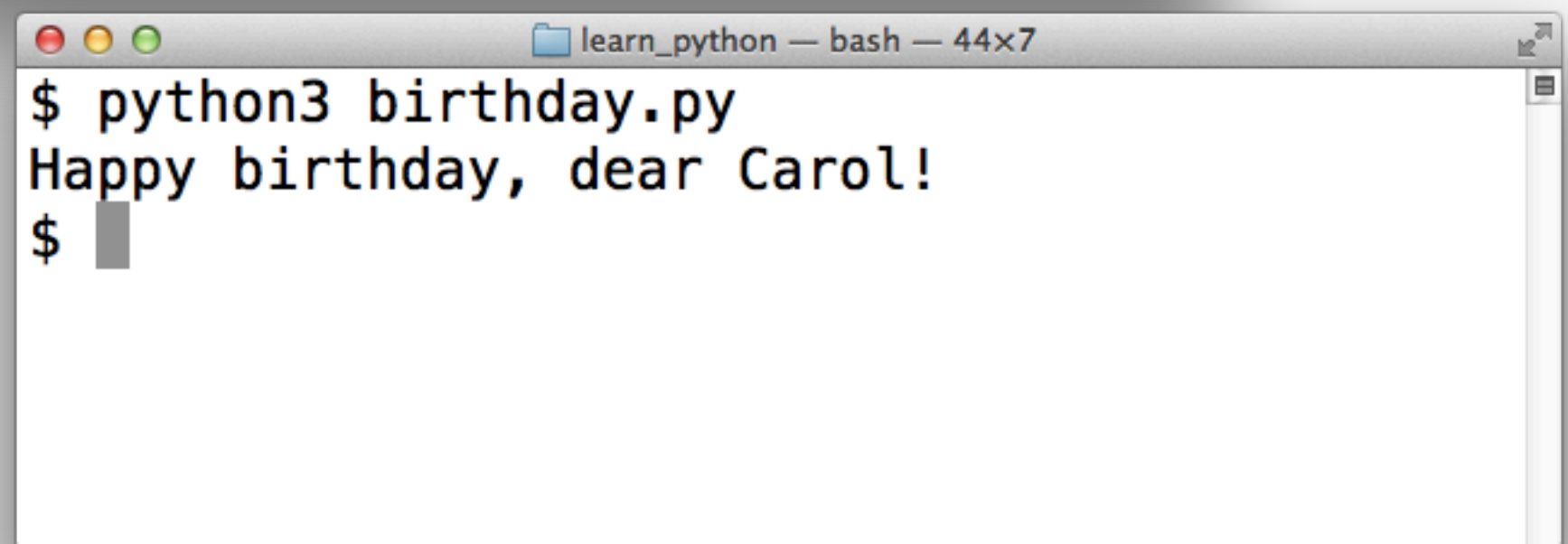
- Happy birthday, dear Carol!

EXERCISE 1: THE ANSWER



```
1 print("Happy birthday, dear Carol!")
2
```

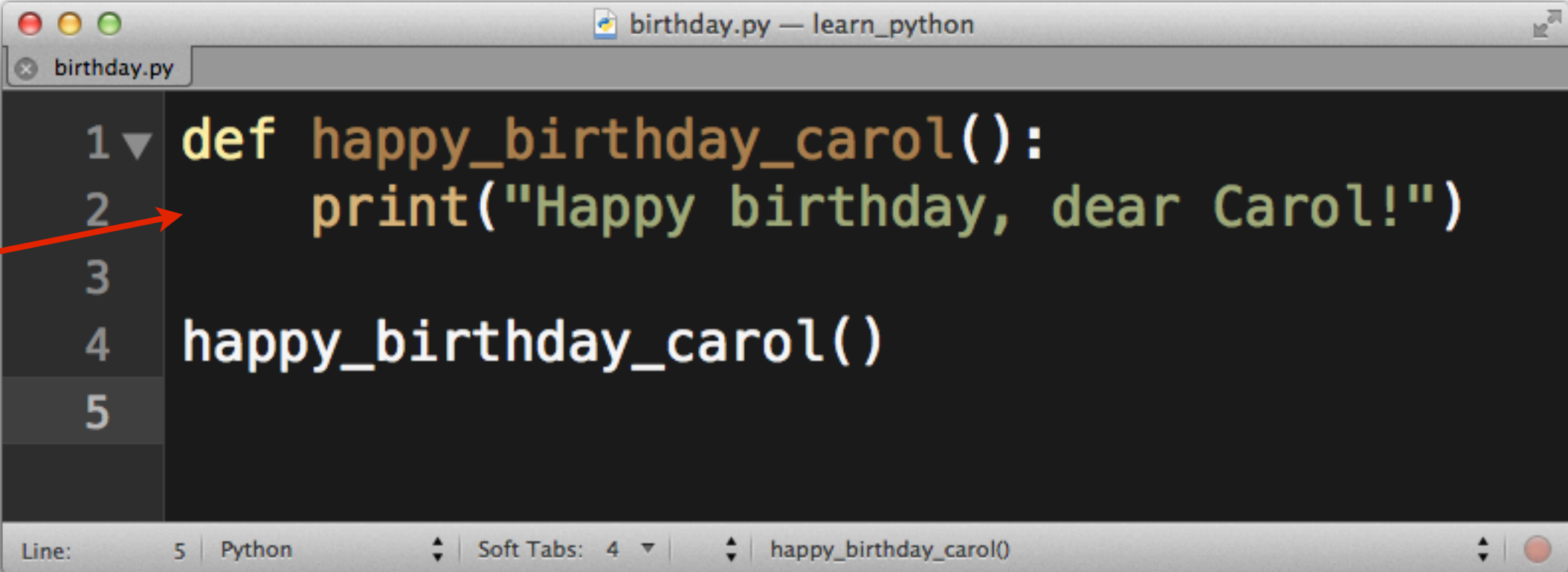
Line: 2 Python Soft Tabs: 4 No symbols to show for current document.



```
$ python3 birthday.py
Happy birthday, dear Carol!
$
```

EXERCISE 2: YOUR CHALLENGE

Change ***birthday.py*** to be this program:



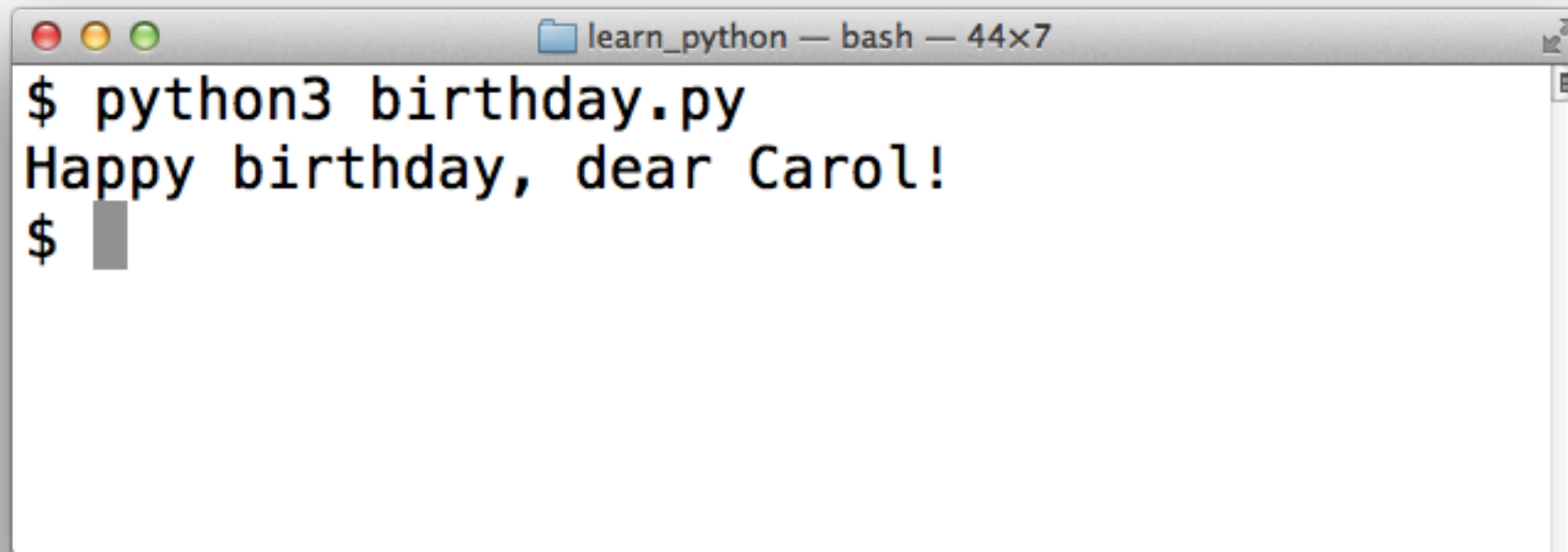
```
1 def happy_birthday_carol():
2     print("Happy birthday, dear Carol!")
3
4 happy_birthday_carol()
5
```

4 spaces
NOT tab

Then run it again to see the same output.

Study the program and try to figure out what's going on.

OUTPUT SHOULD BE THE SAME

A screenshot of a terminal window with a title bar that reads "learn_python — bash — 44x7". The terminal shows a command prompt "\$" followed by the command "python3 birthday.py". The output of the command is "Happy birthday, dear Carol!". Below the output, there is another command prompt "\$" followed by a cursor.

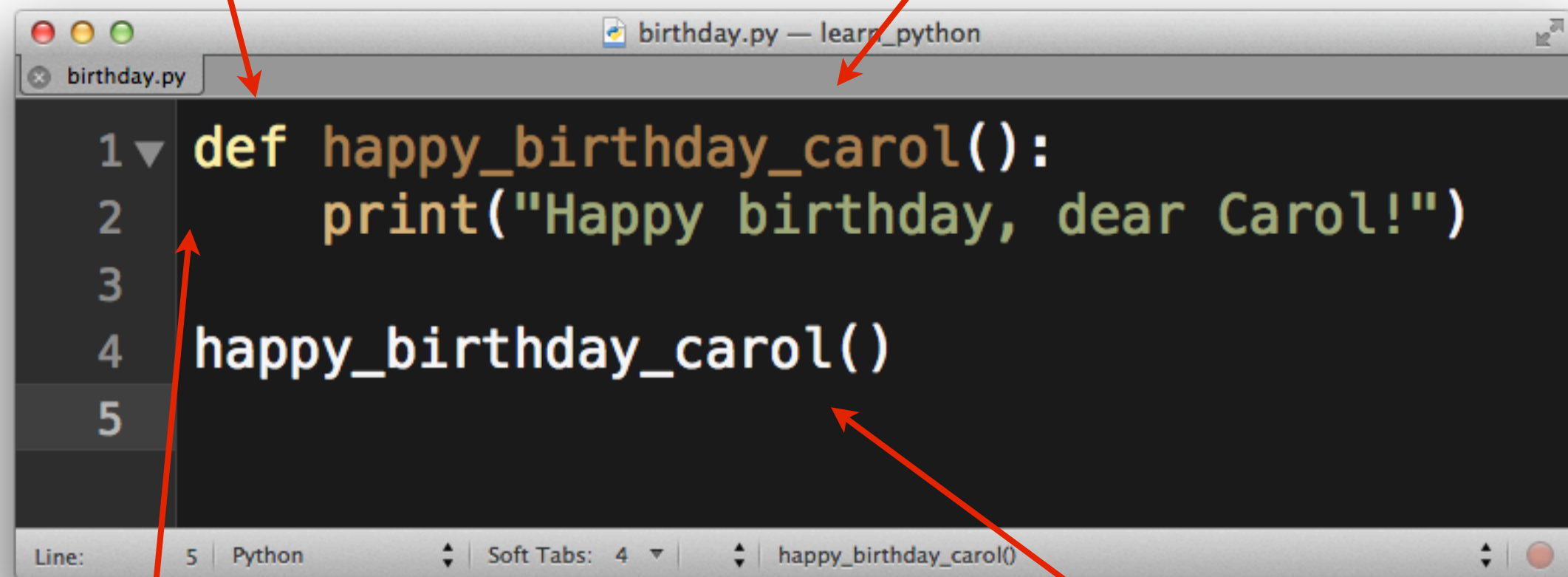
```
$ python3 birthday.py
Happy birthday, dear Carol!
$
```

The program does the same thing, but the code was different this time.

YOU JUST WROTE YOUR FIRST FUNCTION

Indicates start of
function

Name of function



```
1 def happy_birthday_carol():
2     print("Happy birthday, dear Carol!")
3
4 happy_birthday_carol()
5
```

The screenshot shows a code editor window with a tab labeled 'birthday.py'. The code contains a function definition on line 1 and a function call on line 4. Red arrows point from the text 'Indicates start of function' to the 'def' keyword, from 'Name of function' to 'happy_birthday_carol()', from 'Indentation matters' to the indentation of the function body, and from 'Calling the function executes what's inside of it' to the function call.

Indentation
matters

Calling the function executes
what's inside of it

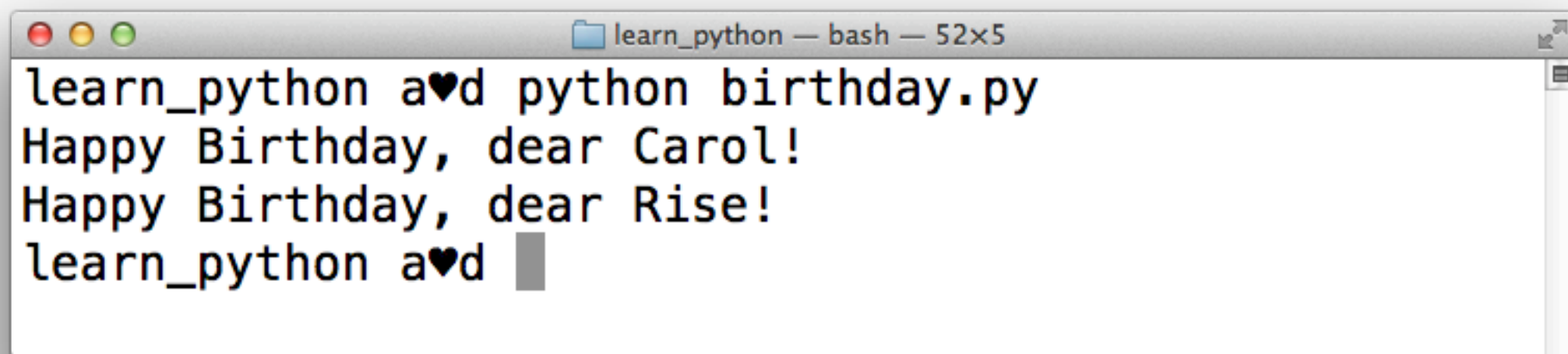
EXERCISE 3: YOUR CHALLENGE

Modify the program to greet *both* Carol and Rise:

- Add a 2nd function to ***birthday.py*** which prints “Happy birthday, dear Rise!”
- Call both functions at the bottom of ***birthday.py***

EXERCISE 3: RECAP

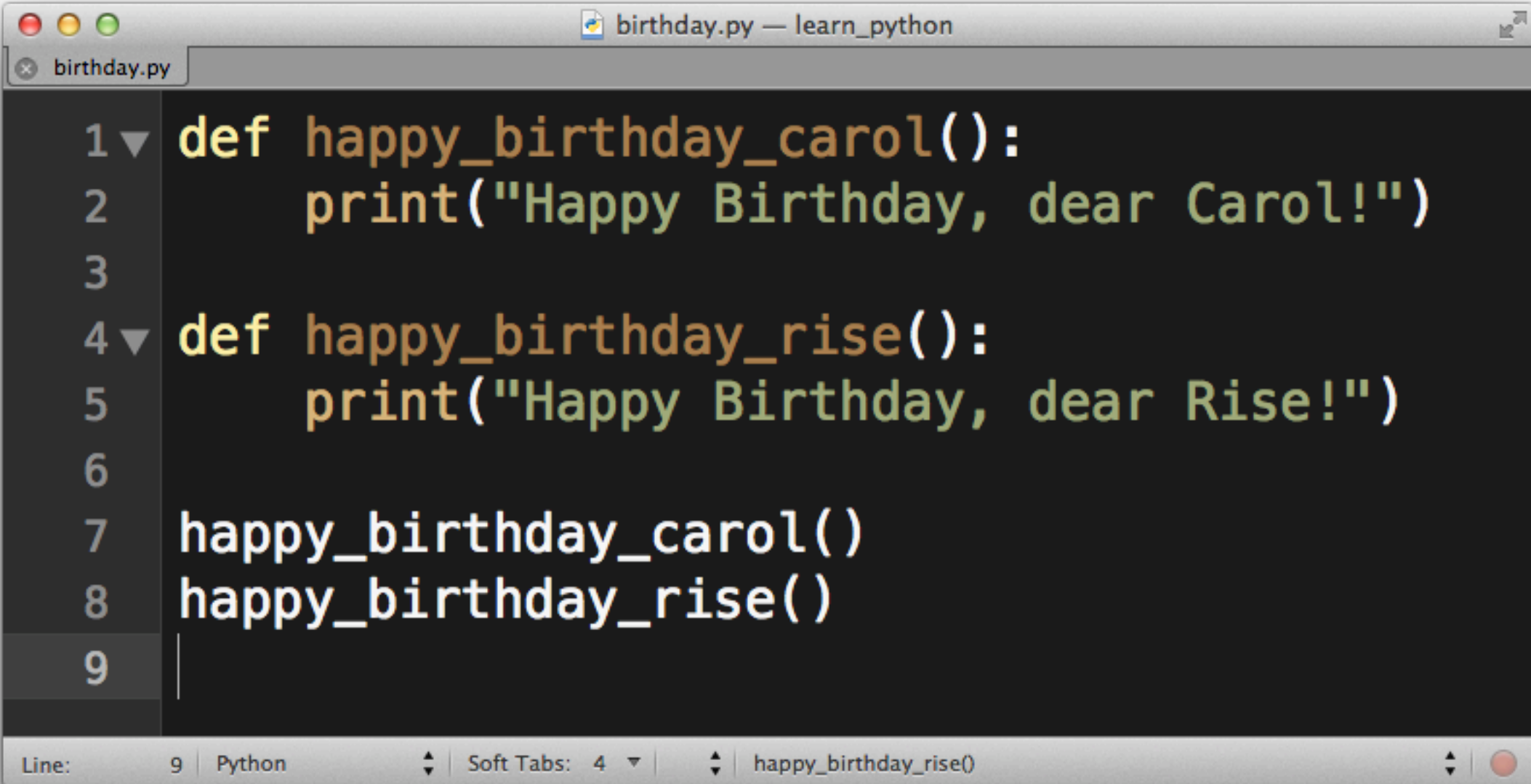
Whenever you run the program, you should see this:

A screenshot of a macOS-style terminal window. The title bar at the top reads 'learn_python — bash — 52x5'. The terminal content shows the command 'learn_python a♥d python birthday.py' being executed. The output consists of two lines: 'Happy Birthday, dear Carol!' and 'Happy Birthday, dear Rise!'. Below the output, the prompt 'learn_python a♥d' is visible with a cursor.

```
learn_python a♥d python birthday.py
Happy Birthday, dear Carol!
Happy Birthday, dear Rise!
learn_python a♥d
```

EXERCISE 3: THE ANSWER

Define 2 functions and call both of them.

A screenshot of a code editor window titled "birthday.py — learn_python". The window shows a Python script with two functions defined and called. The code is as follows:

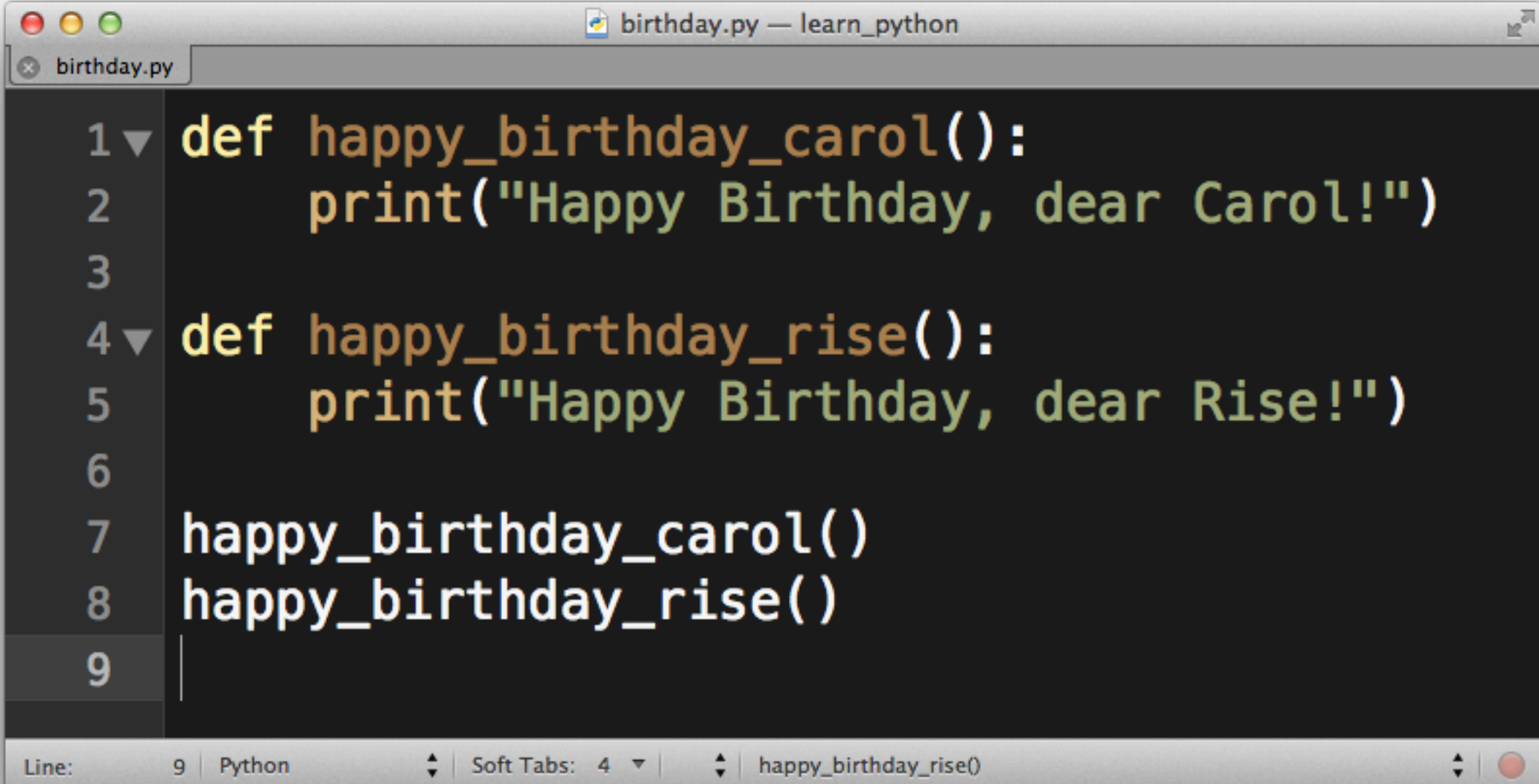
```
1 def happy_birthday_carol():  
2     print("Happy Birthday, dear Carol!")  
3  
4 def happy_birthday_rise():  
5     print("Happy Birthday, dear Rise!")  
6  
7 happy_birthday_carol()  
8 happy_birthday_rise()  
9
```

The editor has a dark theme. The status bar at the bottom shows "Line: 9", "Python", "Soft Tabs: 4", and "happy_birthday_rise()".

```
def happy_birthday_carol():  
    print("Happy Birthday, dear Carol!")  
  
def happy_birthday_rise():  
    print("Happy Birthday, dear Rise!")  
  
happy_birthday_carol()  
happy_birthday_rise()
```

REPETITION?

Notice how this program is repetitive?
We can refactor it.



```
1 def happy_birthday_carol():
2     print("Happy Birthday, dear Carol!")
3
4 def happy_birthday_rise():
5     print("Happy Birthday, dear Rise!")
6
7 happy_birthday_carol()
8 happy_birthday_rise()
9
```

The screenshot shows a code editor window with a dark background. The title bar reads 'birthday.py — learn_python'. The code is written in a light-colored font. Line numbers 1 through 9 are visible on the left. The code defines two functions: 'happy_birthday_carol()' which prints 'Happy Birthday, dear Carol!', and 'happy_birthday_rise()' which prints 'Happy Birthday, dear Rise!'. Both functions are called at the bottom of the file. The status bar at the bottom shows 'Line: 9', 'Python', 'Soft Tabs: 4', and 'happy_birthday_rise()'.

REFACTORING

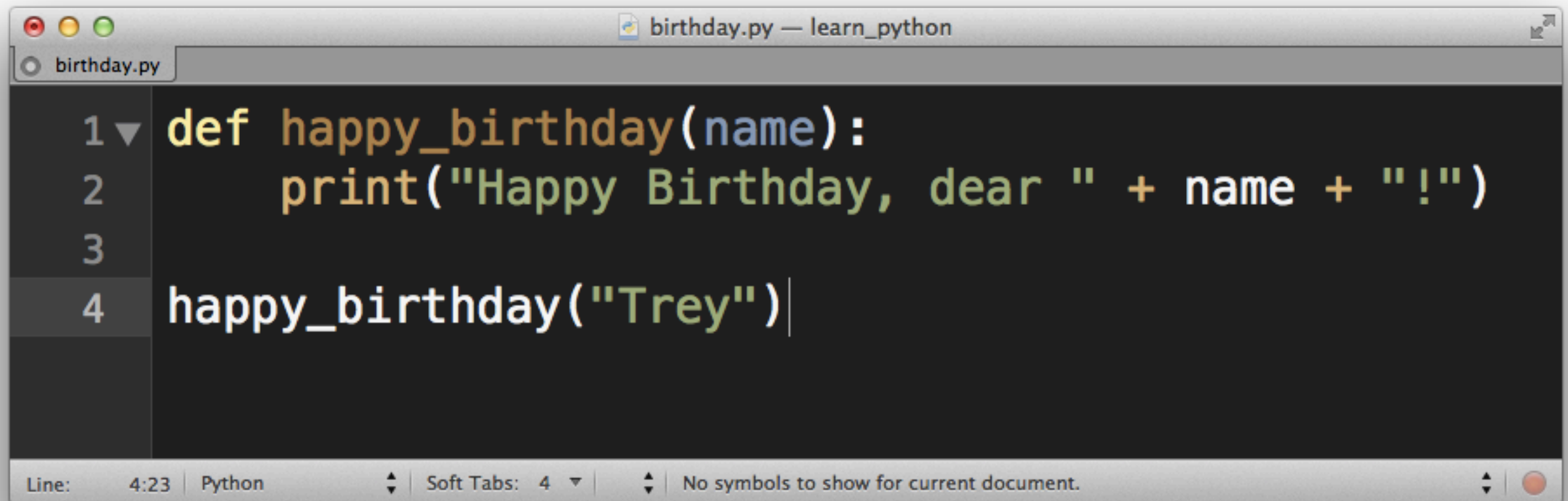
Restructuring code without changing its external behavior

- You did this in Ex 2 without knowing it

We'll now refactor the 2 functions into a single function.

EXERCISE 4

Look carefully at this:



```
1 def happy_birthday(name):  
2     print("Happy Birthday, dear " + name + "!")  
3  
4 happy_birthday("Trey")
```

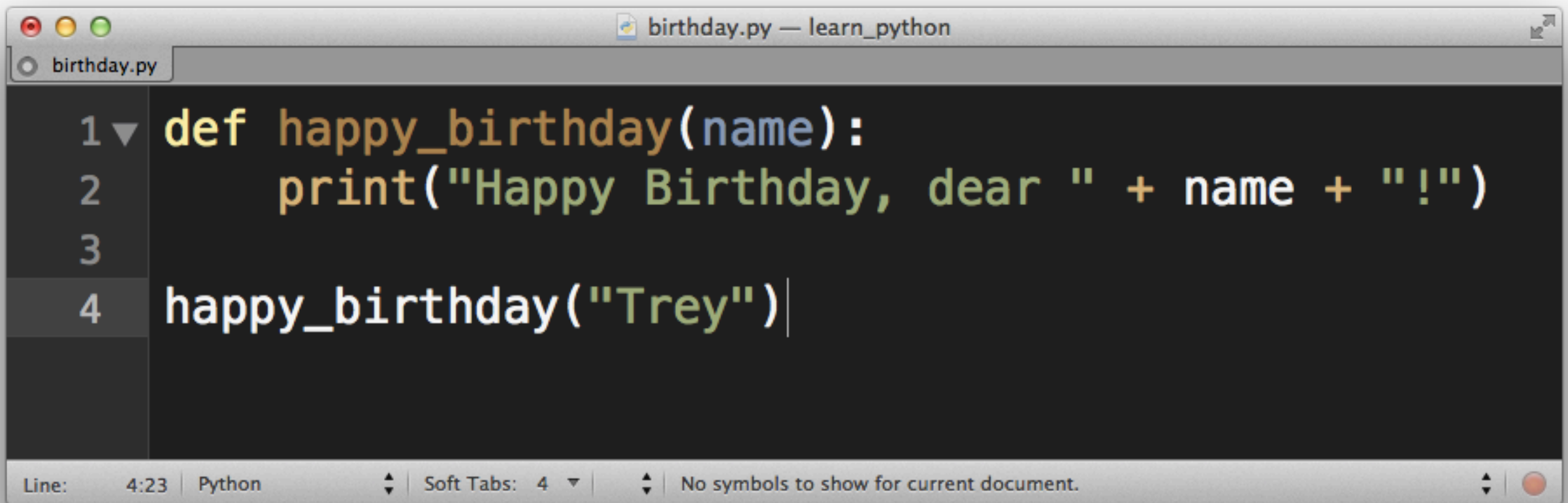
The screenshot shows a code editor window with a title bar that says "birthday.py — learn_python". The editor has a dark background and shows four lines of Python code. Line 1 is a function definition: `def happy_birthday(name):`. Line 2 is an indented print statement: `print("Happy Birthday, dear " + name + "!")`. Line 3 is empty. Line 4 is a function call: `happy_birthday("Trey")`. The cursor is at the end of line 4. The status bar at the bottom shows "Line: 4:23", "Python", "Soft Tabs: 4", and "No symbols to show for current document."

How does it compare to what you currently have in ***birthday.py***?

What's different? What's the same?

EXERCISE 4

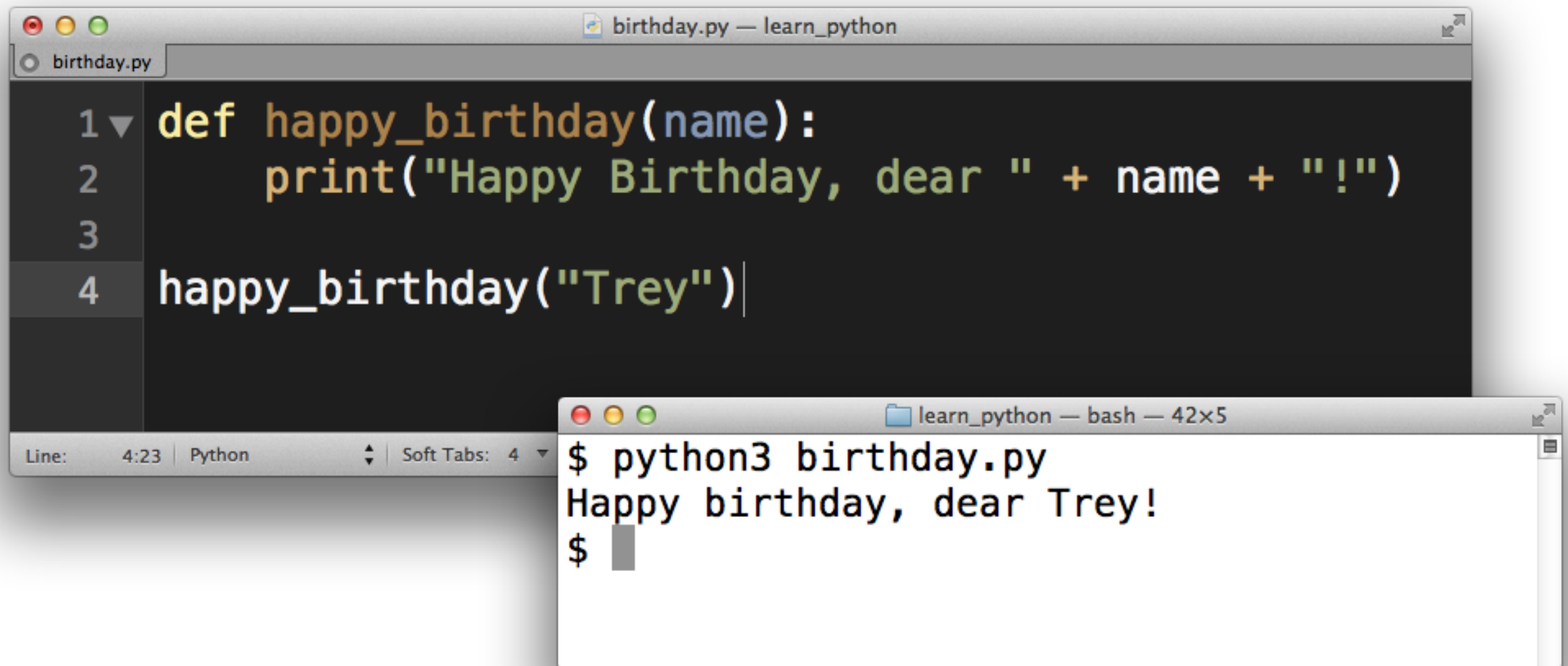
Now, go ahead and replace the text in your ***birthday.py*** file with the text here.

A screenshot of a code editor window titled "birthday.py — learn_python". The editor shows a Python script with four lines of code. Line 1: `def happy_birthday(name):`. Line 2: `print("Happy Birthday, dear " + name + "!!")`. Line 3: (empty line). Line 4: `happy_birthday("Trey")`. The cursor is at the end of line 4. The editor has a dark background with light-colored text. The status bar at the bottom shows "Line: 4:23", "Python", "Soft Tabs: 4", and "No symbols to show for current document.".

```
1 def happy_birthday(name):  
2     print("Happy Birthday, dear " + name + "!!")  
3  
4 happy_birthday("Trey")
```

What do you think will happen when you run it?

EXERCISE 4



The image shows a code editor window titled 'birthday.py — learn_python' with a tab for 'birthday.py'. The code is as follows:

```
1 def happy_birthday(name):  
2     print("Happy Birthday, dear " + name + "!!")  
3  
4 happy_birthday("Trey")
```

Below the code editor is a terminal window titled 'learn_python — bash — 42x5'. It shows the command `$ python3 birthday.py` being executed, resulting in the output `Happy birthday, dear Trey!`. The prompt `$` is shown again on the next line.

The function takes a name argument.
Then it concatenates strings.
Then it prints the final string.

FUNCTION TERMINOLOGY

name is a parameter
(a variable in a function)

This is the
function {

```
1 def happy_birthday(name):
2     print("Happy Birthday, dear " + name + "!")
3
4 happy_birthday("Trey")
```

Calling the
function

"Trey" is an argument
(the value given to name)

EXERCISE 5: YOUR CHALLENGE

Try to break `happy_birthday()` by passing in strange arguments:

- A really, really, really long name
- Numbers
- Other Python types, if you know about any other ones

This is how malicious “black hat” hackers think when they try to break into desktop and web apps.

LONG NAMES: THIS GUY IS REAL

This guy is a real-life test case for name fields

```
happy_birthday("Adolph Blaine Charles David Earl Frederick  
Gerald Hubert Irvin John Kenneth Lloyd Martin Nero Oliver  
Paul Quincy Randolph Sherman Thomas Uncas Victor William  
Xerxes Yancy Zeus Wolfe-schlegelstein-hausenberger-  
dorffvoraltern-waren-gewissenhaft-schaferswessen-  
schafewaren-wohlgepflege-und-sorgfaltigkeit-beschutzen-  
von-angreifen-durch-ihrraubgierigfeinde-welche-voraltern-  
zwolftausend-jahres-vorandieerscheinen-wander-ersteer-dem-  
enschderraumschiff-gebrauchlicht-als-sein-ursprung-von-  
kraftgestart-sein-lange-fahrt-hinzwischen-sternartigraum-  
auf-der-suchenach-diestern-welche-gehabt-bewohnbar-  
planeten-kreise-drehen-sich-und-wohin-der-neurasse-von-  
verstandigmen-schlichkeit-konnte-fortpflanzen-und-sicher-  
freuen-anlebens-langlich-freude-und-ruhe-mit-nicht-ein-  
furcht-vor-angreifen-von-anderer-intelligent-geschopfs-  
von-hinzwischen-sternartigraum, Senior")
```

EXERCISE 6

Numbers when strings are expected:




```
1 def hi(first, last):  
2     print("Hi " + first + last)  
3  
4 hi("Henry", 8)
```

Line: 4:15 Python Soft Tabs: 4 hi(first, last)

Oops! Passing in 8 instead of a string makes this fail.

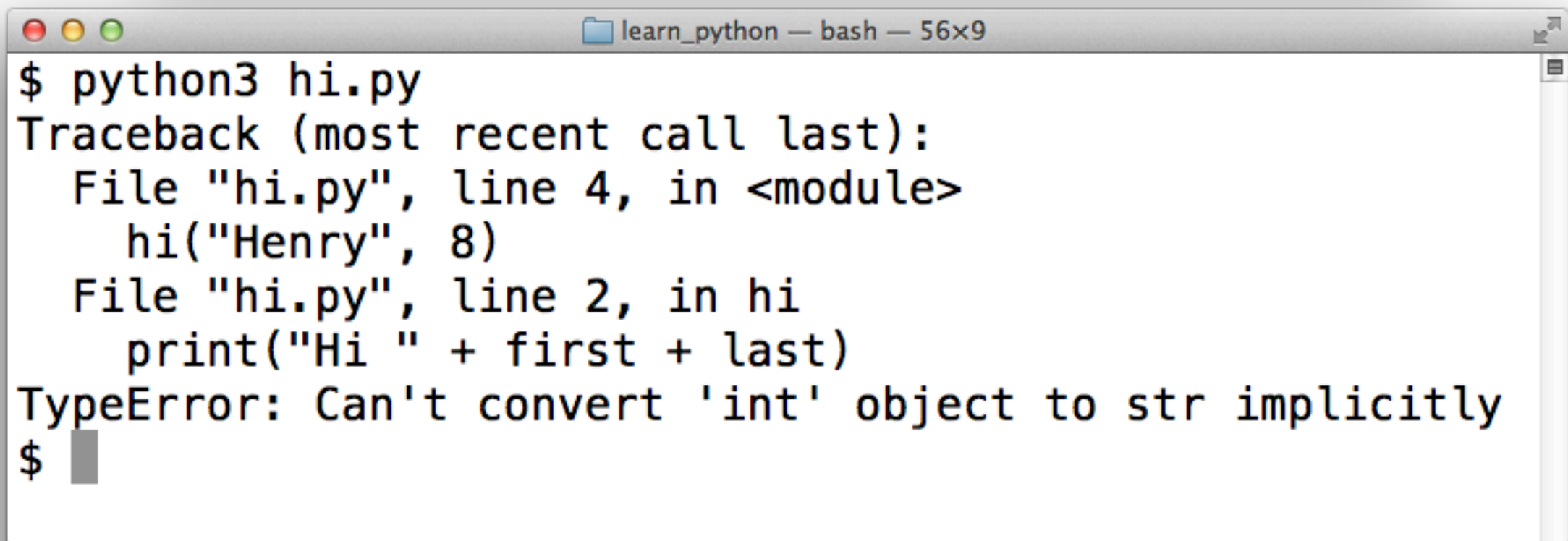
Can anyone guess why?

EXERCISE 6: ANSWER



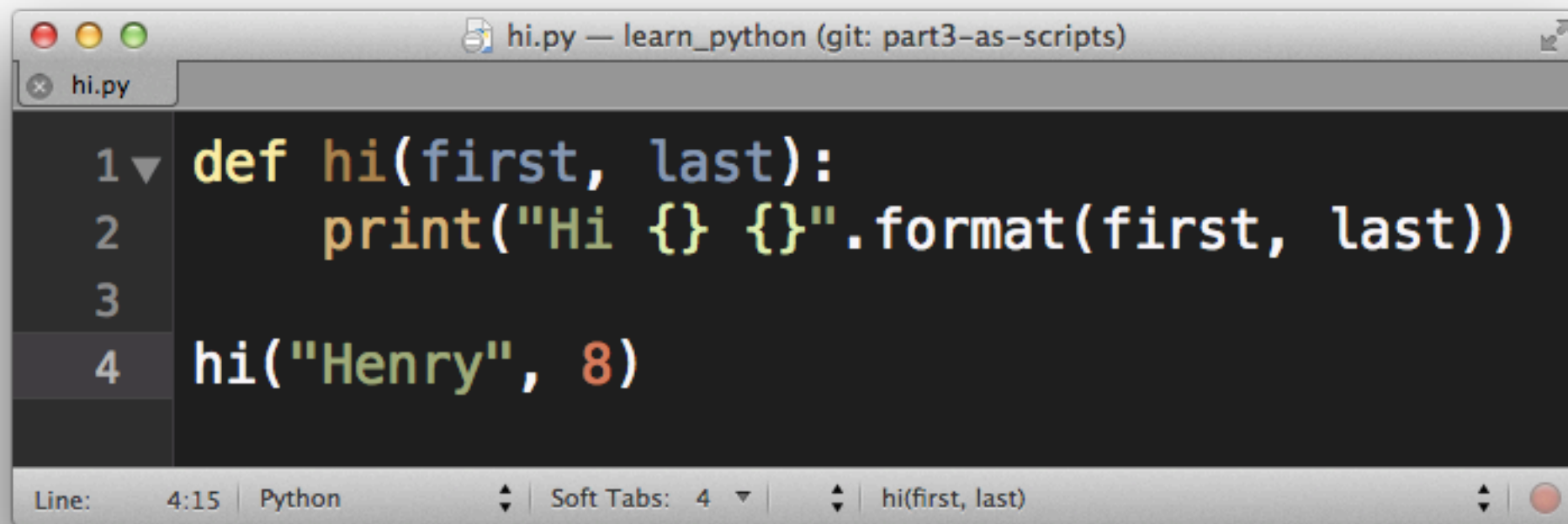
```
1 def hi(first, last):
2     print("Hi " + first + last)
3
4 hi("Henry", 8)
```

Line: 4:15 Python Soft Tabs: 4 hi(first, last)



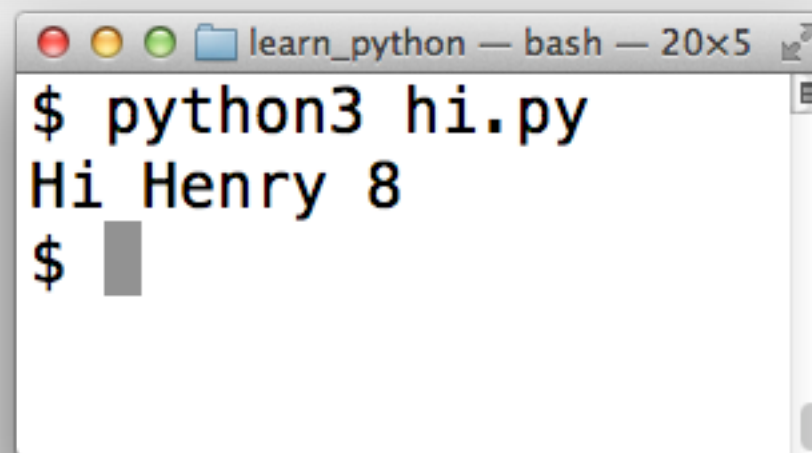
```
$ python3 hi.py
Traceback (most recent call last):
  File "hi.py", line 4, in <module>
    hi("Henry", 8)
  File "hi.py", line 2, in hi
    print("Hi " + first + last)
TypeError: Can't convert 'int' object to str implicitly
$
```

EXERCISE 7: STRING FORMATTING



```
hi.py — learn_python (git: part3-as-scripts)
1 def hi(first, last):
2     print("Hi {} {}".format(first, last))
3
4 hi("Henry", 8)
```

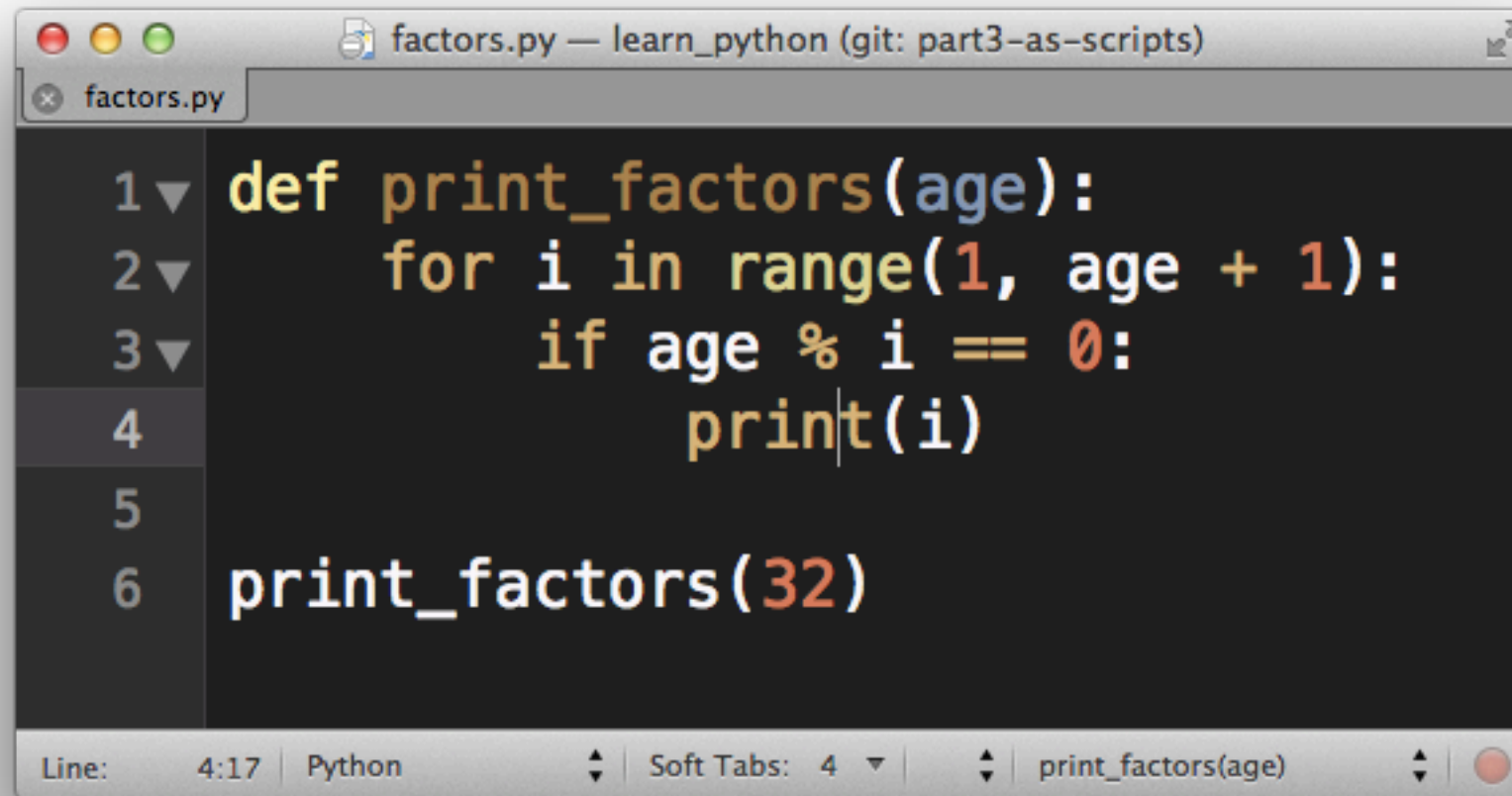
Line: 4:15 Python Soft Tabs: 4 hi(first, last)



```
learn_python — bash — 20x5
$ python3 hi.py
Hi Henry 8
$
```

`str.format()` is **not** the same as string concatenating!

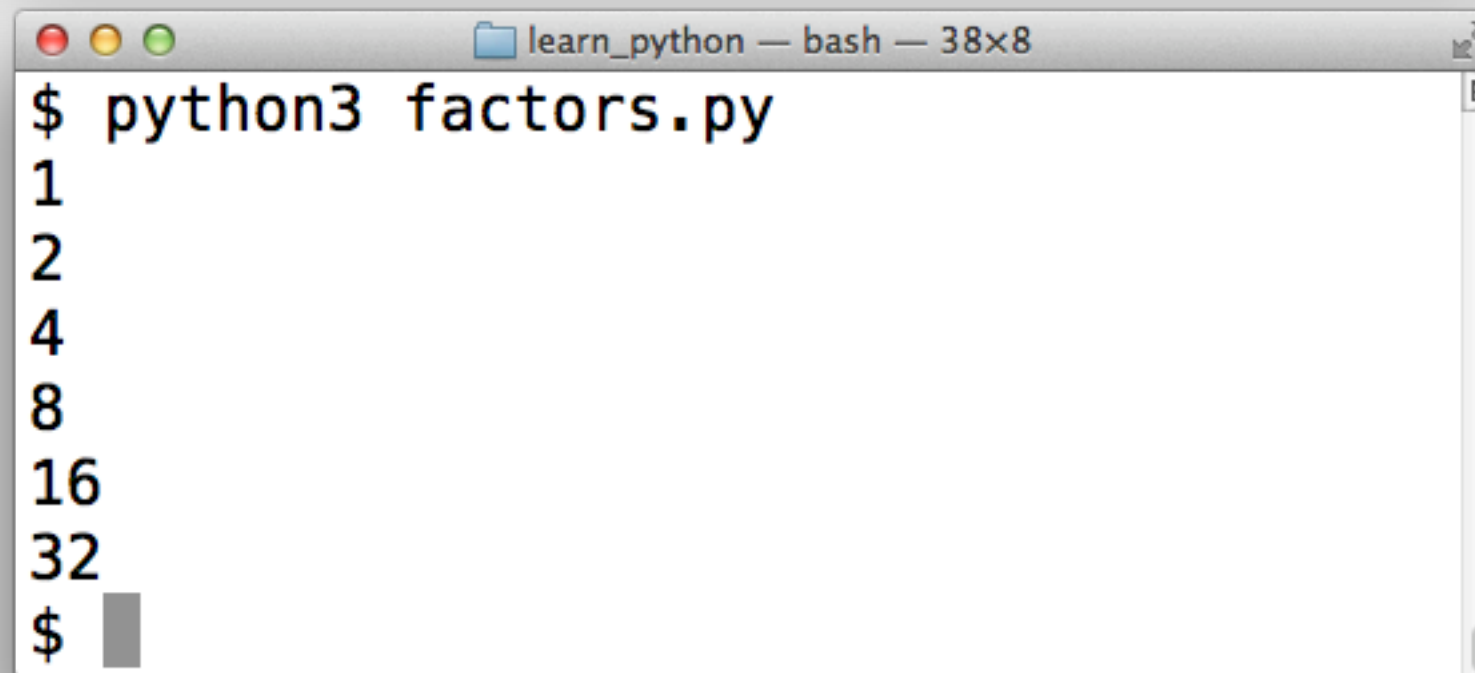
EXERCISE 8: FANCY MATH



A screenshot of a code editor window titled "factors.py — learn_python (git: part3-as-scripts)". The editor shows a Python script with the following code:

```
1 def print_factors(age):  
2     for i in range(1, age + 1):  
3         if age % i == 0:  
4             print(i)  
5  
6 print_factors(32)
```

The status bar at the bottom indicates "Line: 4:17", "Python", "Soft Tabs: 4", and "print_factors(age)".



A screenshot of a terminal window titled "learn_python — bash — 38x8". The terminal shows the command `$ python3 factors.py` and its output:

```
$ python3 factors.py  
1  
2  
4  
8  
16  
32  
$
```

EXERCISE 9: CUPCAKE TALLY

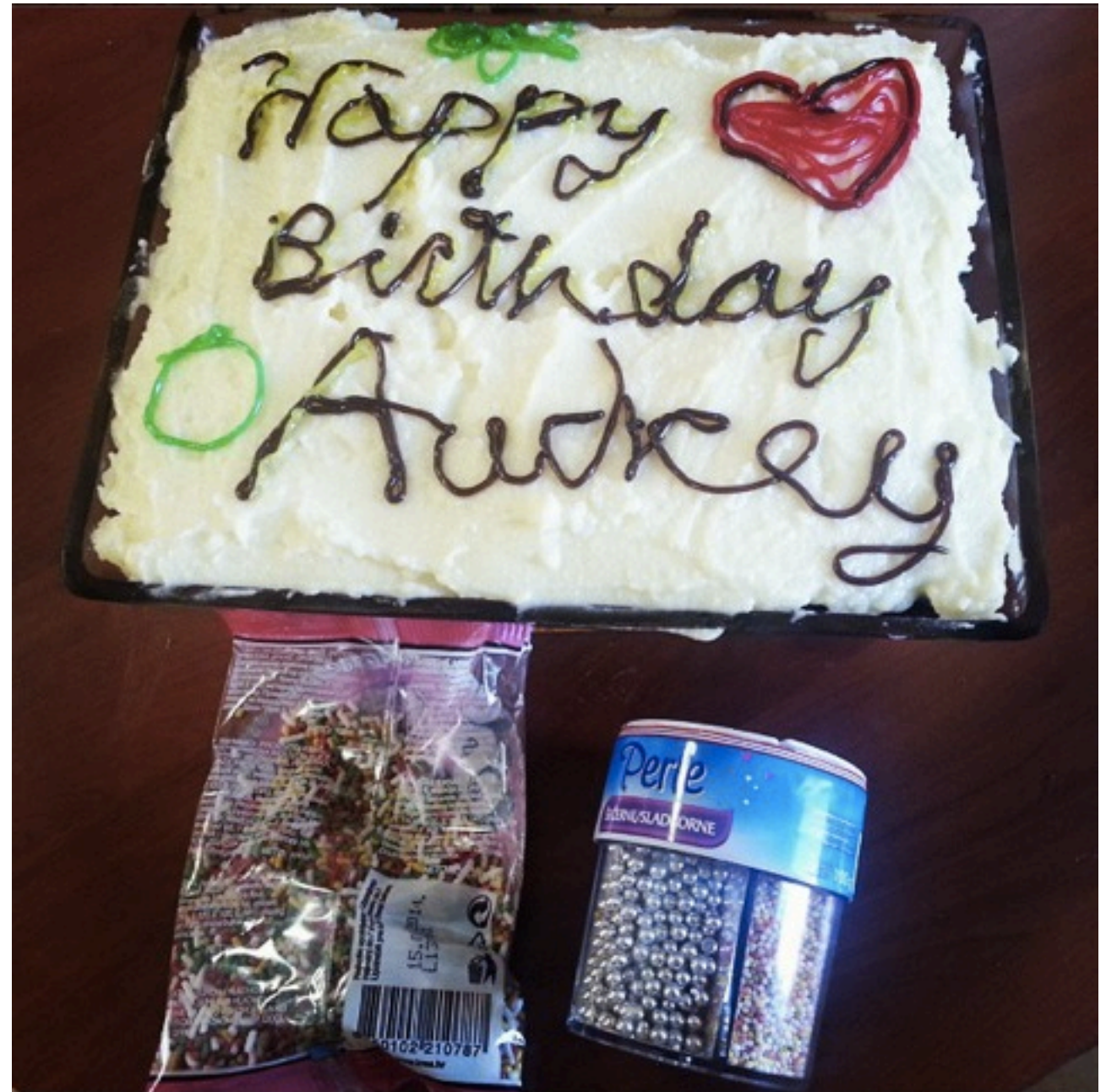
A good birthday party should allocate:

- 2 cupcakes per guest
- 13 cupcakes for the birthday person, because if it's your birthday, you should be allowed to eat a baker's dozen of cupcakes.

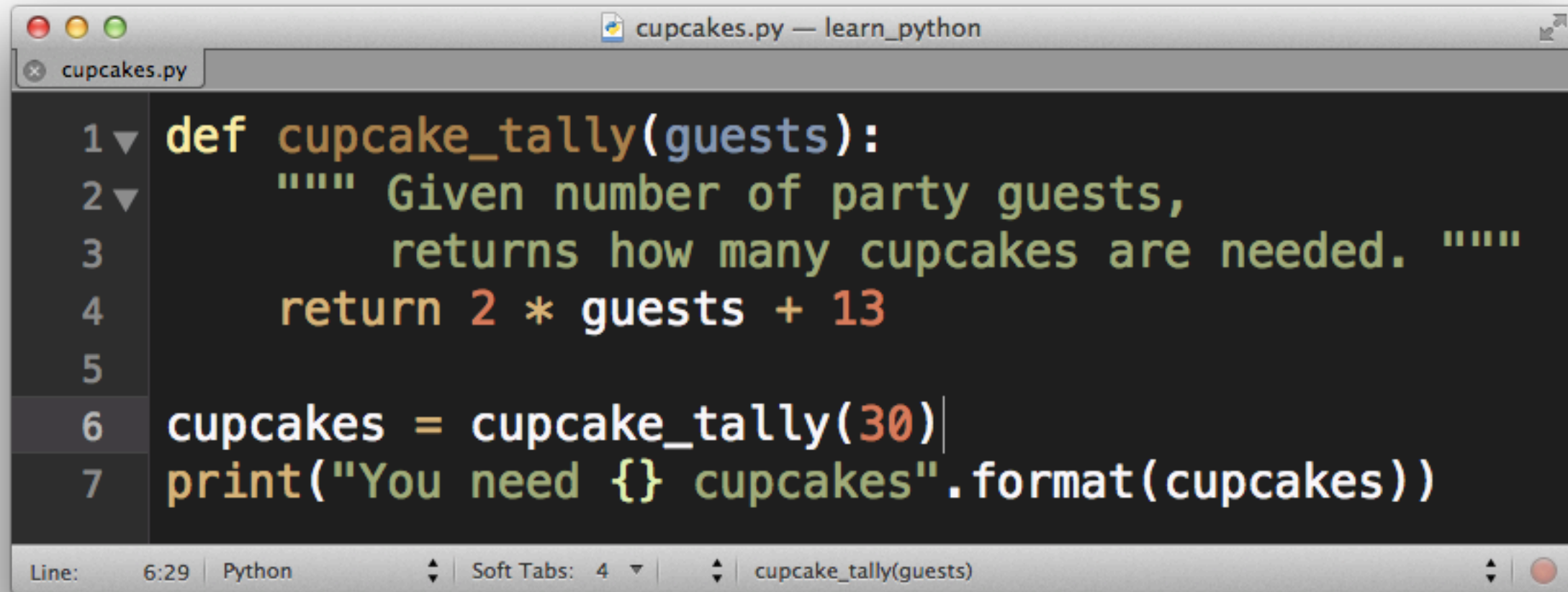
That's how I like to plan things :)

EXERCISE 9: CUPCAKE TALLY

Normally the math is too confusing, so I stick with cake



EXERCISE 9: CUPCAKE TALLY



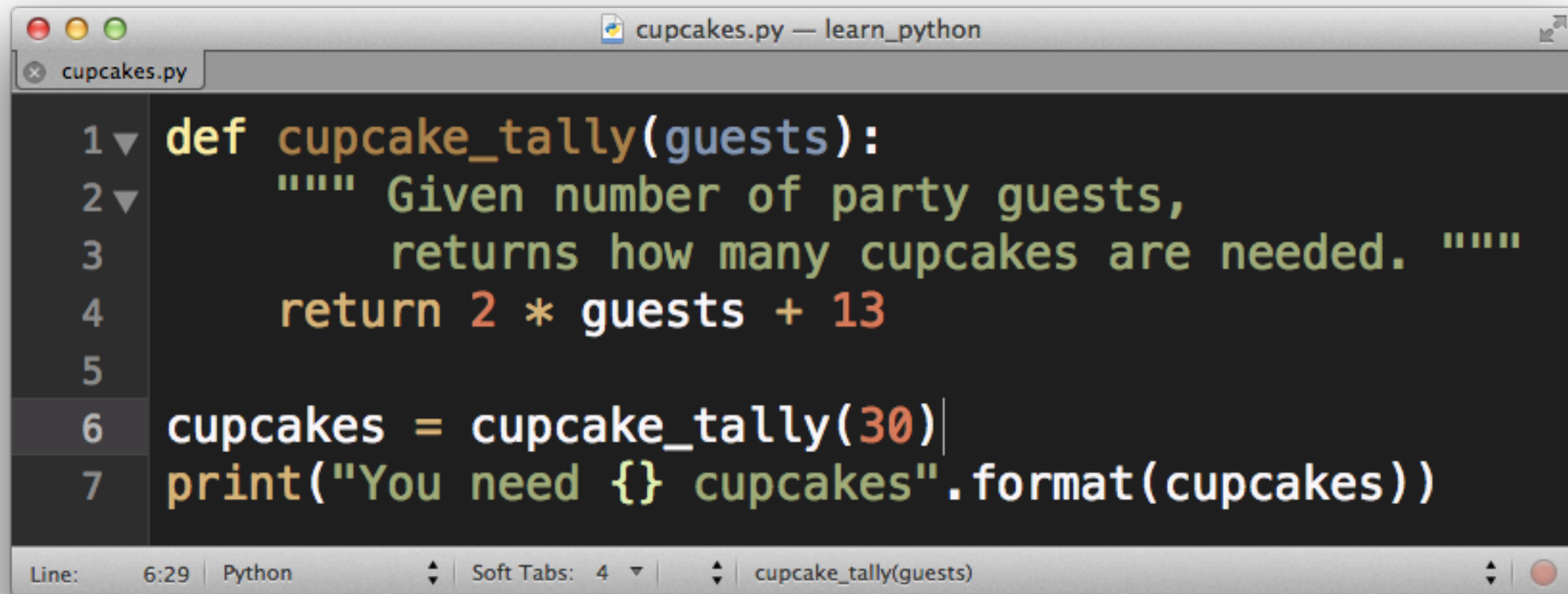
```
1 def cupcake_tally(guests):
2     """ Given number of party guests,
3         returns how many cupcakes are needed. """
4     return 2 * guests + 13
5
6 cupcakes = cupcake_tally(30)
7 print("You need {} cupcakes".format(cupcakes))
```

Line: 6:29 Python Soft Tabs: 4 cupcake_tally(guests)

A good birthday party should allocate:

- 2 cupcakes per guest
- 13 cupcakes for the birthday person, because if it's your birthday, you should be allowed to eat a baker's dozen of cupcakes.

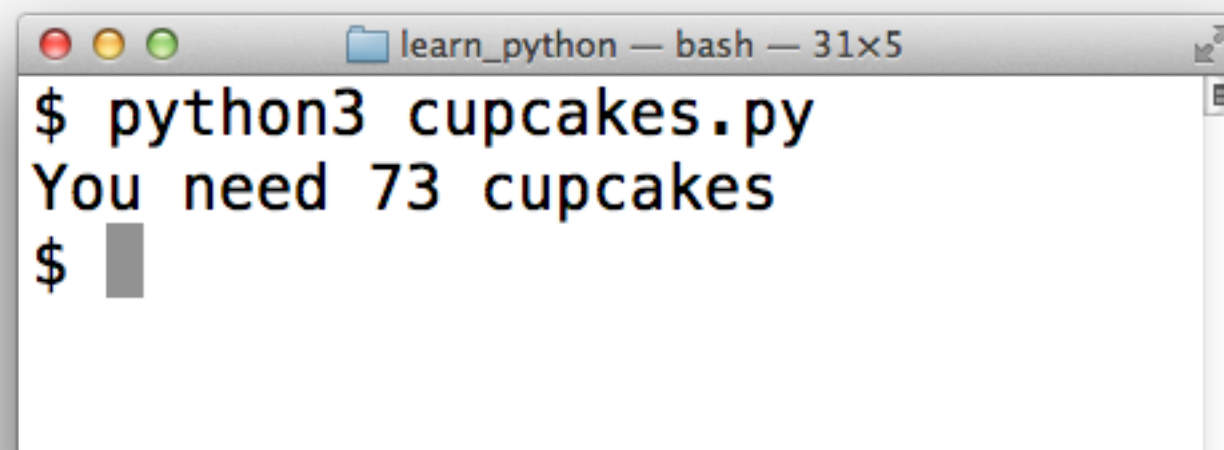
EXERCISE 9: CUPCAKE TALLY



```
cupcakes.py — learn_python
cupcakes.py
1 def cupcake_tally(guests):
2     """ Given number of party guests,
3         returns how many cupcakes are needed. """
4     return 2 * guests + 13
5
6 cupcakes = cupcake_tally(30)
7 print("You need {} cupcakes".format(cupcakes))

Line: 6:29 Python Soft Tabs: 4 cupcake_tally(guests)
```

cupcake_tally returns a value which we can print



```
learn_python — bash — 31x5
$ python3 cupcakes.py
You need 73 cupcakes
$
```

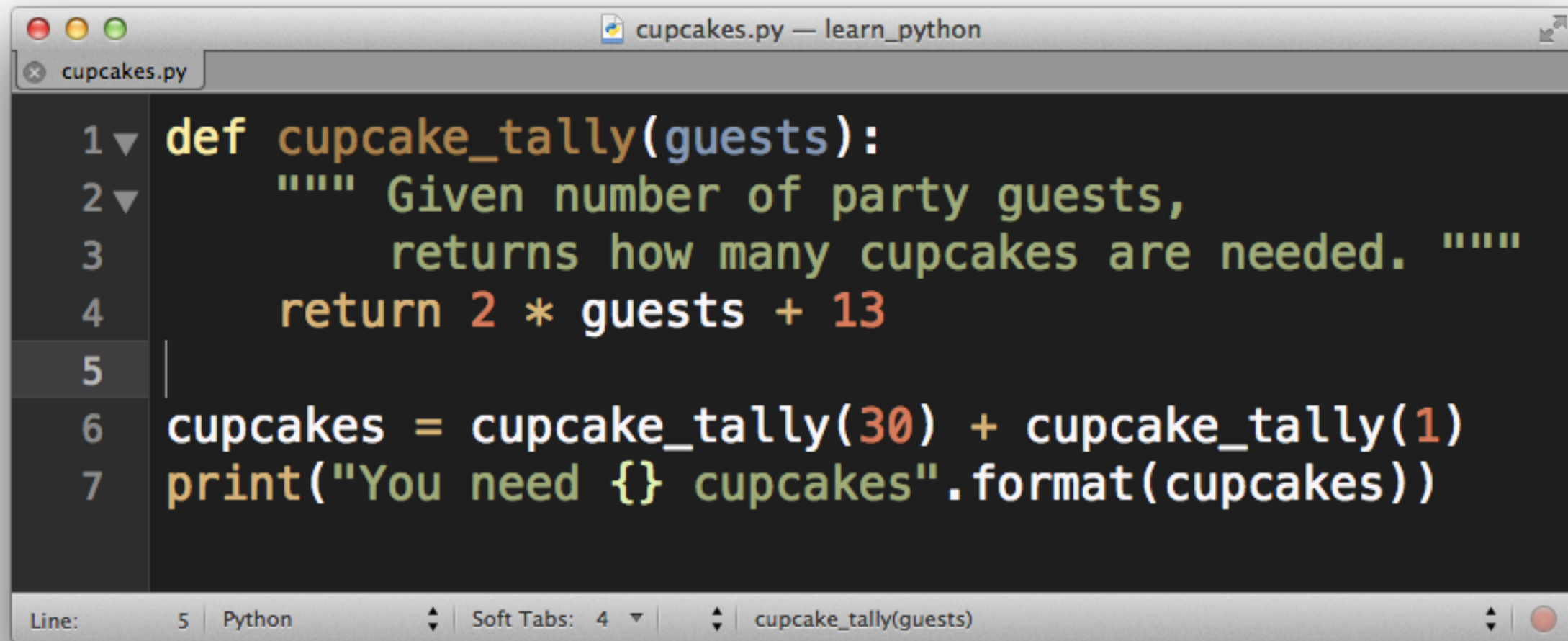
EXERCISE 9: CUPCAKE TALLY

What about 2 parties:

- My 30-guest party tonight
- My private party with just Daniel tomorrow morning

How many cupcakes would we need to bake?

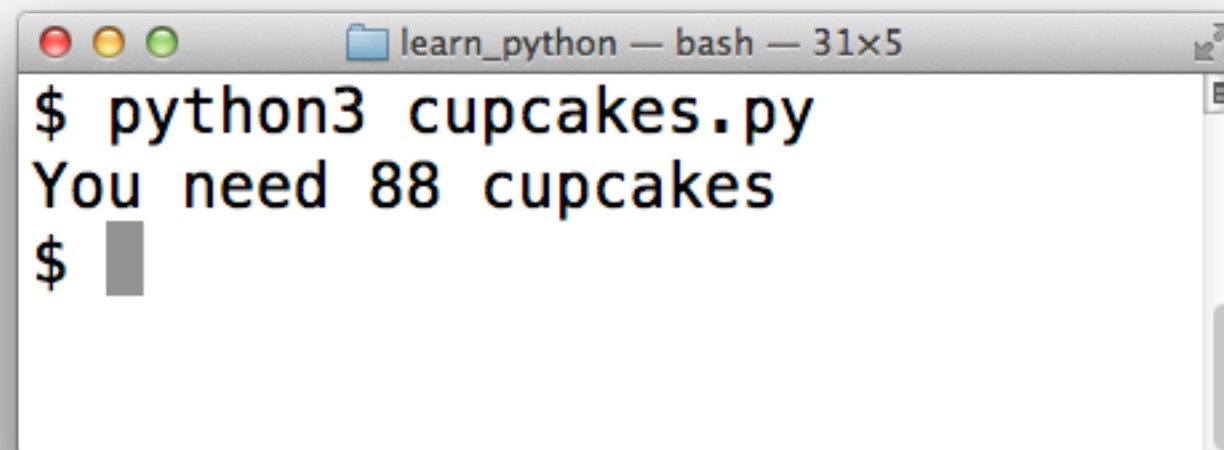
EXERCISE 9: CUPCAKE TALLY



```
1 def cupcake_tally(guests):  
2     """ Given number of party guests,  
3         returns how many cupcakes are needed. """  
4     return 2 * guests + 13  
5  
6 cupcakes = cupcake_tally(30) + cupcake_tally(1)  
7 print("You need {} cupcakes".format(cupcakes))
```

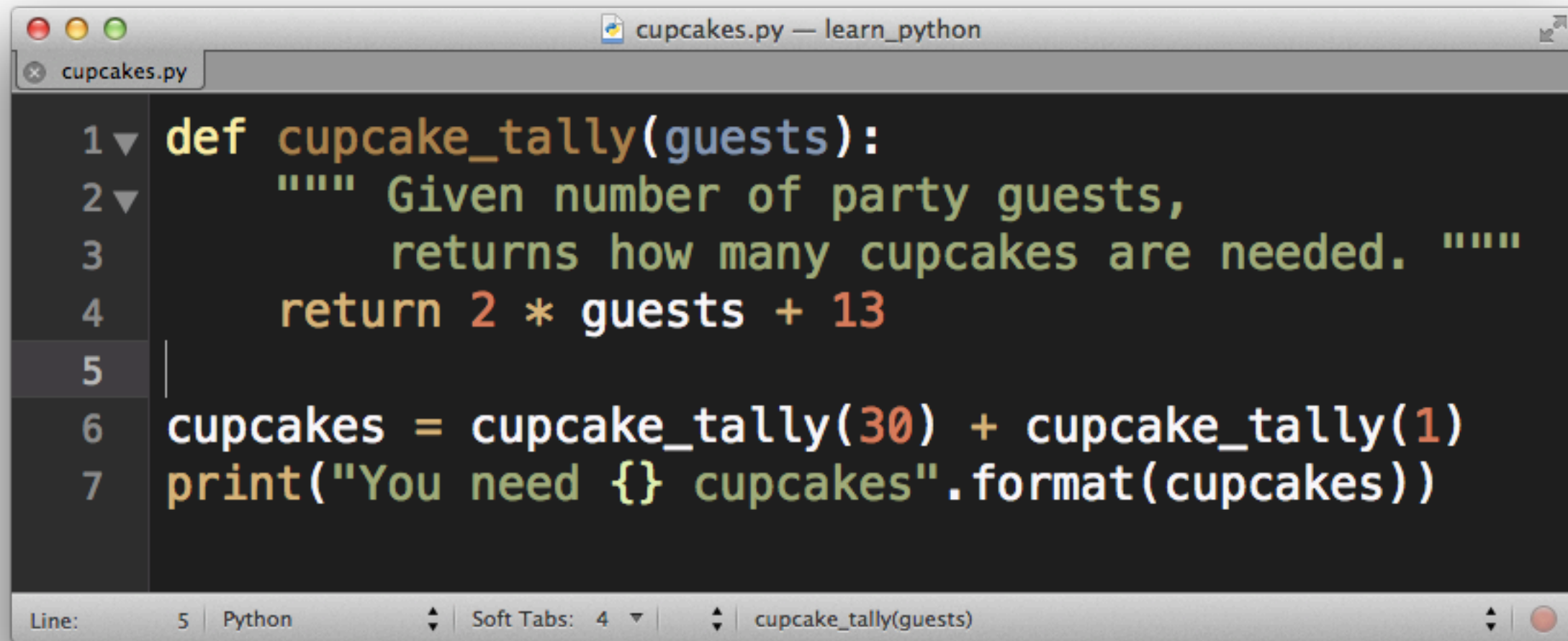
Line: 5 Python Soft Tabs: 4 cupcake_tally(guests)

Call the function twice!



```
learn_python — bash — 31x5  
$ python3 cupcakes.py  
You need 88 cupcakes  
$
```

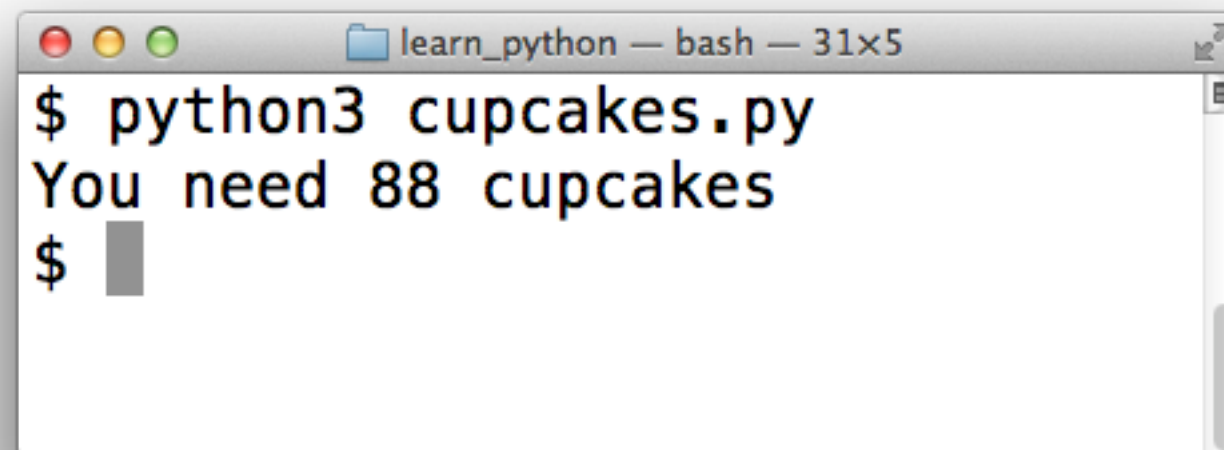
EXERCISE 9: CUPCAKE TALLY



```
1 def cupcake_tally(guests):
2     """ Given number of party guests,
3         returns how many cupcakes are needed. """
4     return 2 * guests + 13
5
6 cupcakes = cupcake_tally(30) + cupcake_tally(1)
7 print("You need {} cupcakes".format(cupcakes))
```

The screenshot shows a code editor window titled 'cupcakes.py — learn_python'. The code defines a function 'cupcake_tally' that takes 'guests' as an argument and returns the number of cupcakes needed based on the formula $2 \times \text{guests} + 13$. Below the function definition, the script calls 'cupcake_tally(30)' and 'cupcake_tally(1)', adds their results, and prints the total using a formatted string.

We add the results of both `cupcake_tally()` calls.



```
$ python3 cupcakes.py
You need 88 cupcakes
$
```

The screenshot shows a terminal window titled 'learn_python — bash — 31x5'. It displays the command '\$ python3 cupcakes.py' being executed, followed by the output 'You need 88 cupcakes' and a new prompt '\$'.

SUMMARY:

WHAT ARE FUNCTIONS USED FOR?

Program decomposition, or factoring

To break down a problem into parts.

It makes the code:

- More readable.
- Easier to understand.

SUMMARY:

WHAT ARE FUNCTIONS USED FOR?

Code reuse

Functions can be used instead of:

- Repeating the same lines of code at different times throughout a program.

This reduces duplication of code.

SUMMARY:

WHAT ARE FUNCTIONS USED FOR?

Abstraction or simplification

Using functions allows us to:

- Hide all of the details involved
- Put them into one place
- Simply call the function to execute the lines of code.

MODULES

For grouping reusable code like:

- Functions
- Important constants like `MAX_CUPCAKES`
- Other useful stuff, e.g. decorators, context managers

Put your functions into modules for later reuse!

EXERCISE 10: MODULES

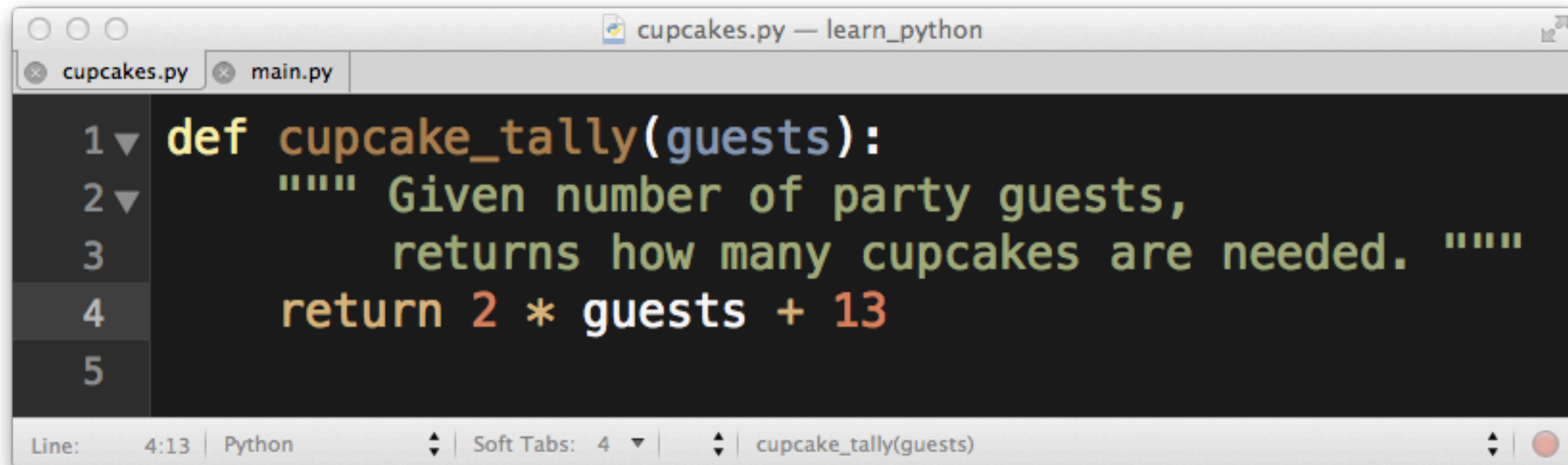
Break our code into 2 files for reuse:

- ***cupcakes.py*** holds the function(s)
- ***main.py*** is the main program

(.py files are called ‘modules’)

EXERCISE 10: MODULES

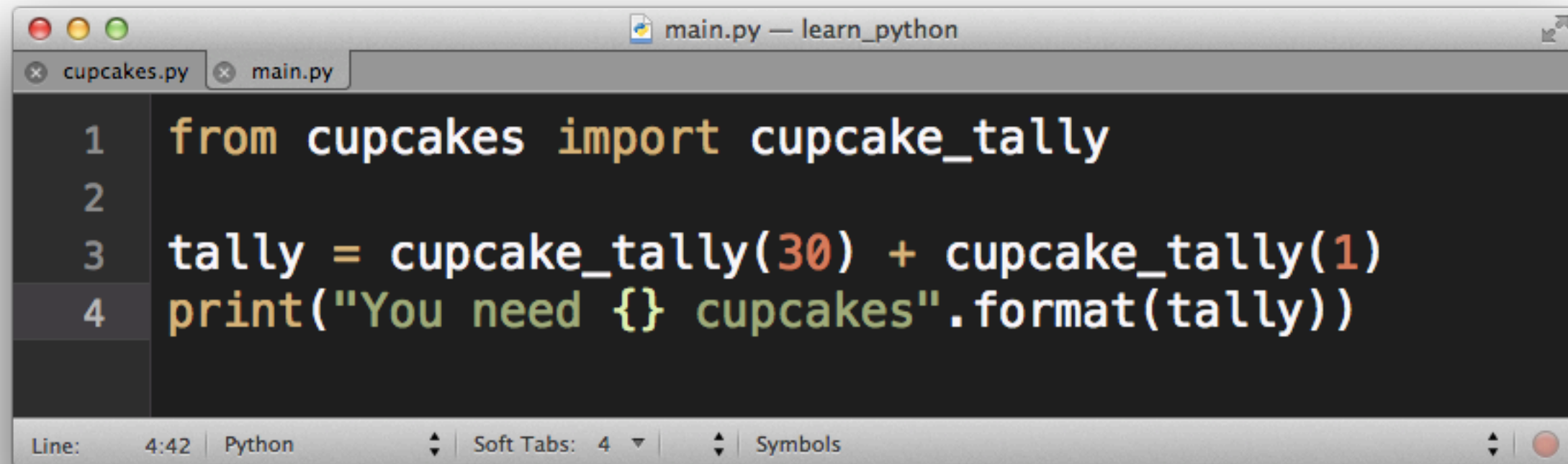
cupcakes.py



```
1 def cupcake_tally(guests):  
2     """ Given number of party guests,  
3         returns how many cupcakes are needed. """  
4     return 2 * guests + 13  
5
```

Line: 4:13 Python Soft Tabs: 4 cupcake_tally(guests)

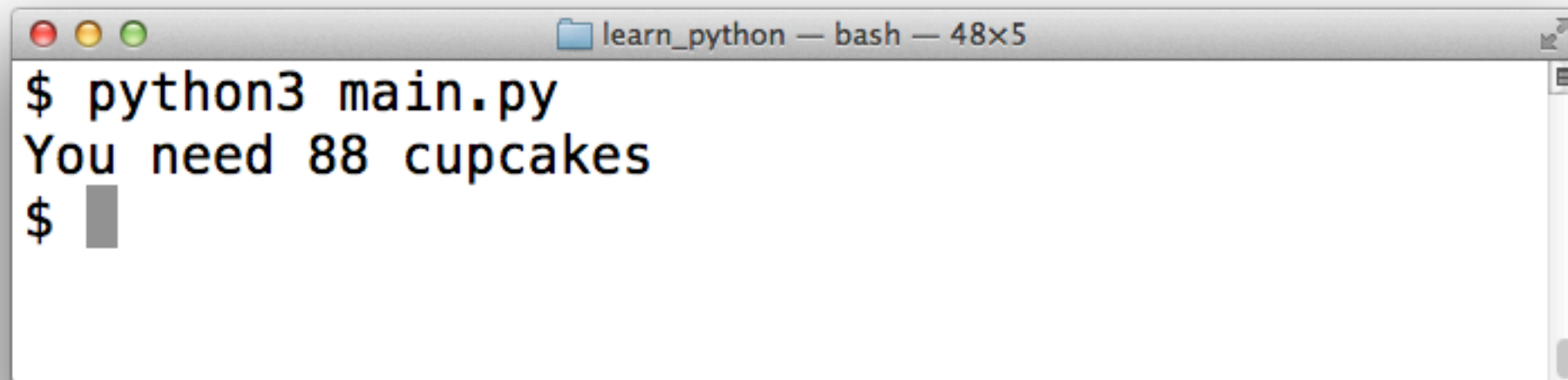
main.py



```
1 from cupcakes import cupcake_tally  
2  
3 tally = cupcake_tally(30) + cupcake_tally(1)  
4 print("You need {} cupcakes".format(tally))
```

Line: 4:42 Python Soft Tabs: 4 Symbols

EXERCISE 10: MODULES

A terminal window titled 'learn_python — bash — 48x5' with standard macOS window controls. It shows the command '\$ python3 main.py' being executed, followed by the output 'You need 88 cupcakes' and a new prompt '\$' with a cursor.

```
$ python3 main.py
You need 88 cupcakes
$
```

- ***main.py*** imports `cupcake_tally` from ***cupcakes.py***
- ***main.py*** calls the `cupcake_tally` function twice
- ***main.py*** sums the returned value of both function calls and prints the result.

WAIT, THERE'S MORE!

You can also find and use modules from:

- Python standard library (stdlib)
- Python Package Index (PyPI)

I'll cover those at the end of the day, in the conclusion.